Introduction to Indexing
Tree-Structured Indexes

Storage – Layers seen so far

- Hardware
- Disk Space Manager
- Buffer Manager
- A layer that provides the abstraction of a file of records (records == tuples)
  - May be implemented on top of pages in a variety of ways

Next question

- How to organize records within files (logically) for good performance?
- Depends on how the records will be accessed in query processing

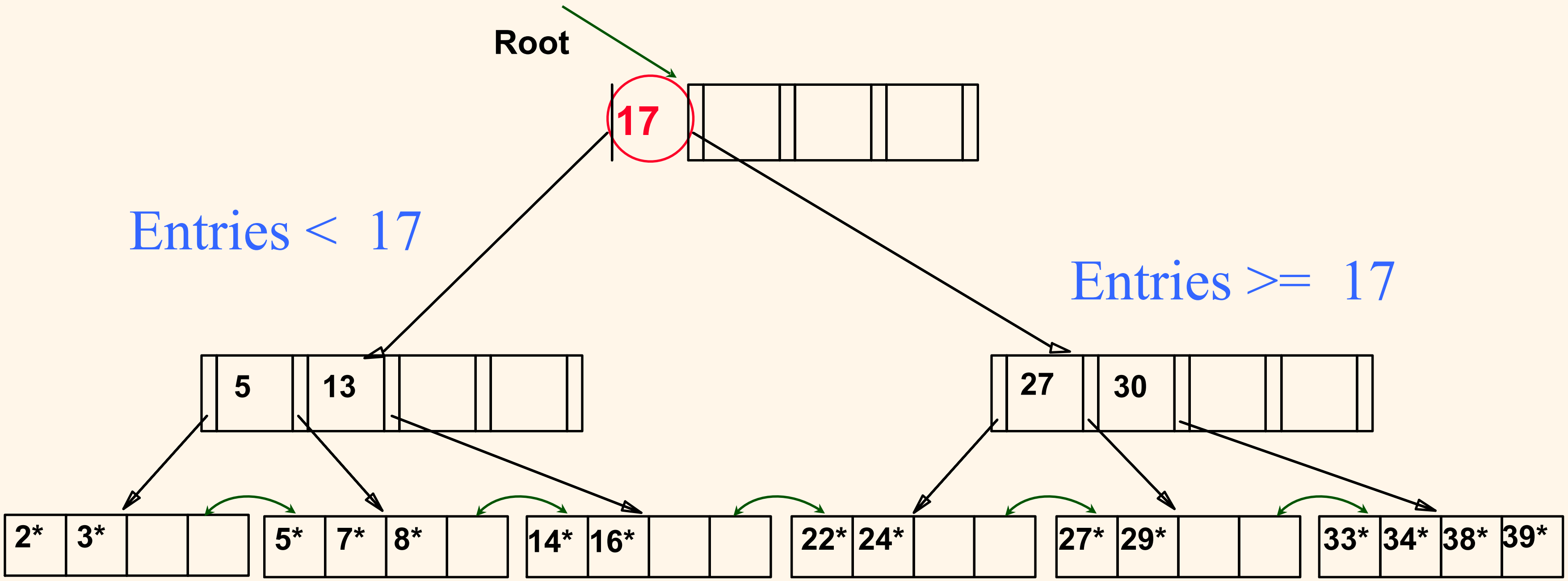
Some Possible File Organizations

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a "range" of records is needed.
  - But, can only sort on one attribute!
  - How to maintain order when inserting
- Indexes: Data structures to support efficient retrieval of records via trees or hashing.

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation)

Example Tree Index



Indexes

- An index contains a collection of *data entries*
- A data entry **k*** is information that allows retrieval of all records with a given key value **k**.

Alternatives for Data Entry k in Index*

- In a data entry k* we can store:
  - Data record with key value **k,** or
  - <**k**, rid of data record with search key value **k**>, or
  - <**k**, list of rids of data records with search key **k**>
- Could use each of these alternatives with a tree index, or a hash index, etc.

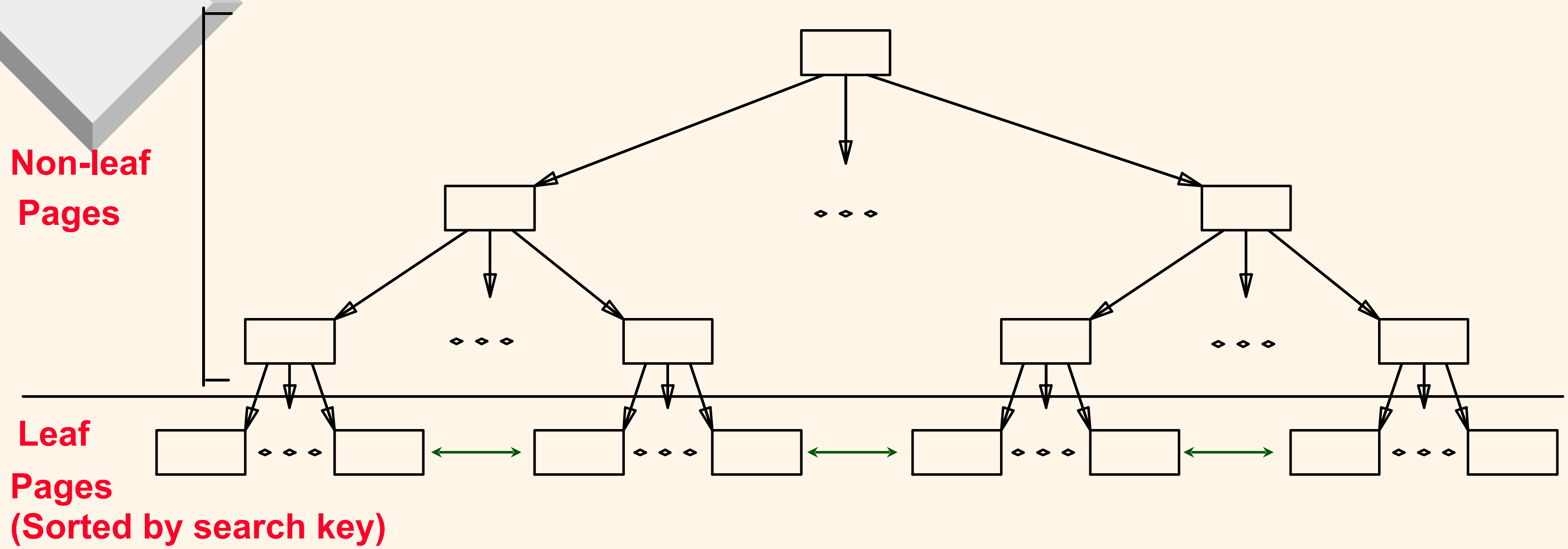
Alternatives for Data Entries (Contd.)

- Alternative 1:
  - If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).
  - At most one index on a given collection of data records should use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)

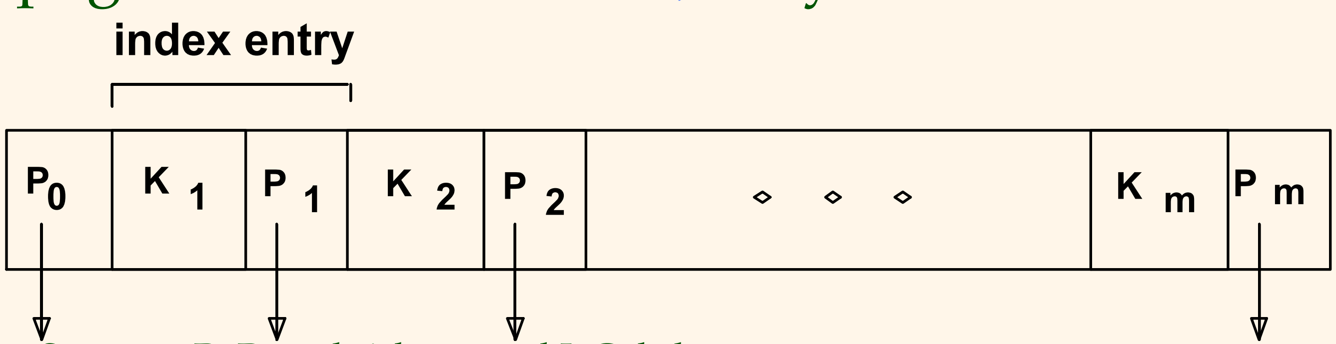
Alternatives for Data Entries (Contd.)

- Alternatives 2 and 3:
  - Alternative 3 more compact than Alternative 2, and no duplicate keys in index
  - But Alternative 3 leads to variable sized data entries even if search keys are of fixed length.

B+ Tree Indexes



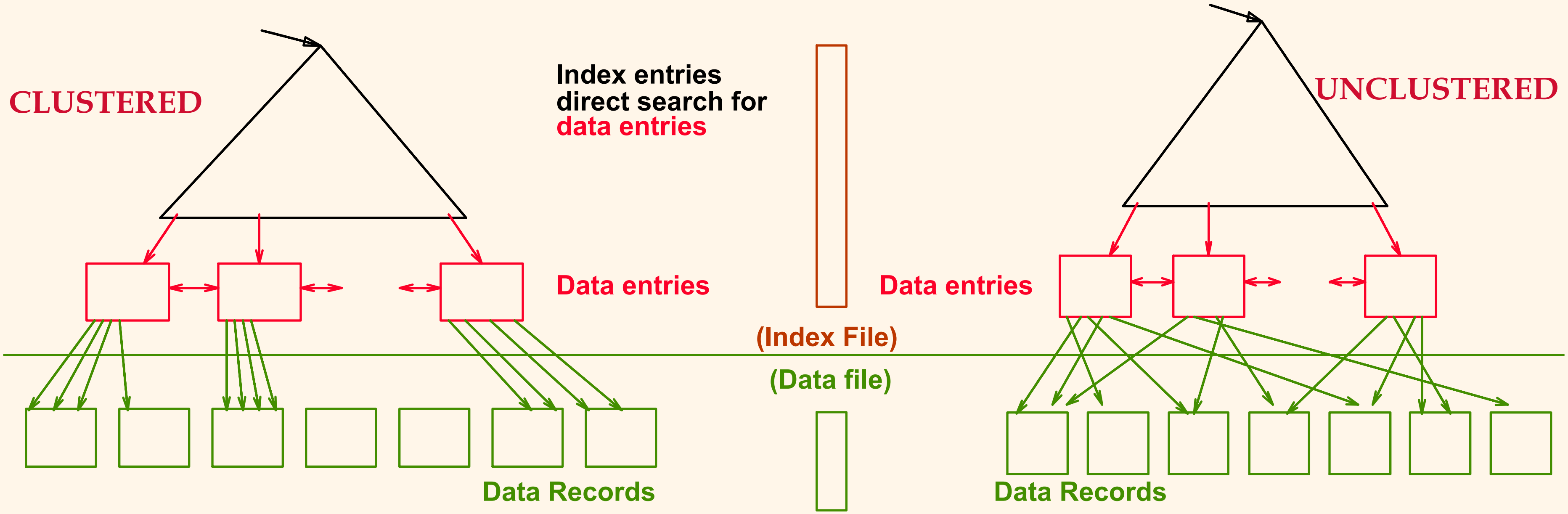
- Leaf pages contain *data entries*
- Non-leaf pages have *index entries;* only used to direct searches:



Clustered vs unclustered indexes

- If order of data records is the same as, or "close to", order of data entries, then called clustered index.

Clustered vs. Unclustered Index



Clustered vs unclustered indexes

- If order of data records is the same as, or "close to", order of data entries, then called clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Composite search keys

- A search key for indexing may contain more than one attribute
  - E.g. can have index on <name, age>
- Ordering on search keys induced by orderings on respective attributes
  - <Alice, 30> comes before <Alice, 50>, which in comes before <Bob, 10>

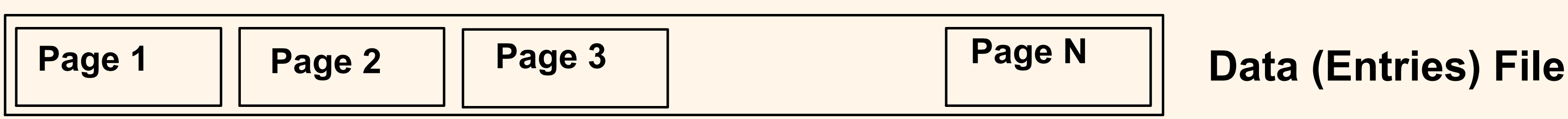
Tree-structured indexing

- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

- Simple cost metric for discussion of search costs: number of disk I/Os (i.e. how many pages need to be brought in from disk)
  - Ignore benefits of sequential access etc to simplify

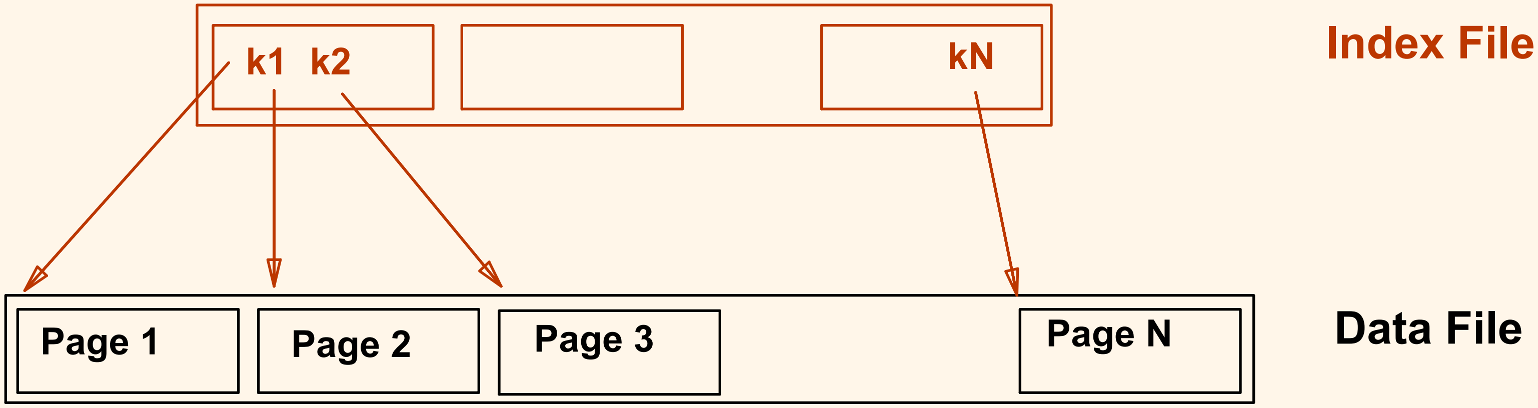
Range Searches

- ``*Find all students with gpa > 3.0*''
  - If data *entries* are sorted, do binary search to find first such student, then scan to find others.
- Cost of this search?



Range Searches

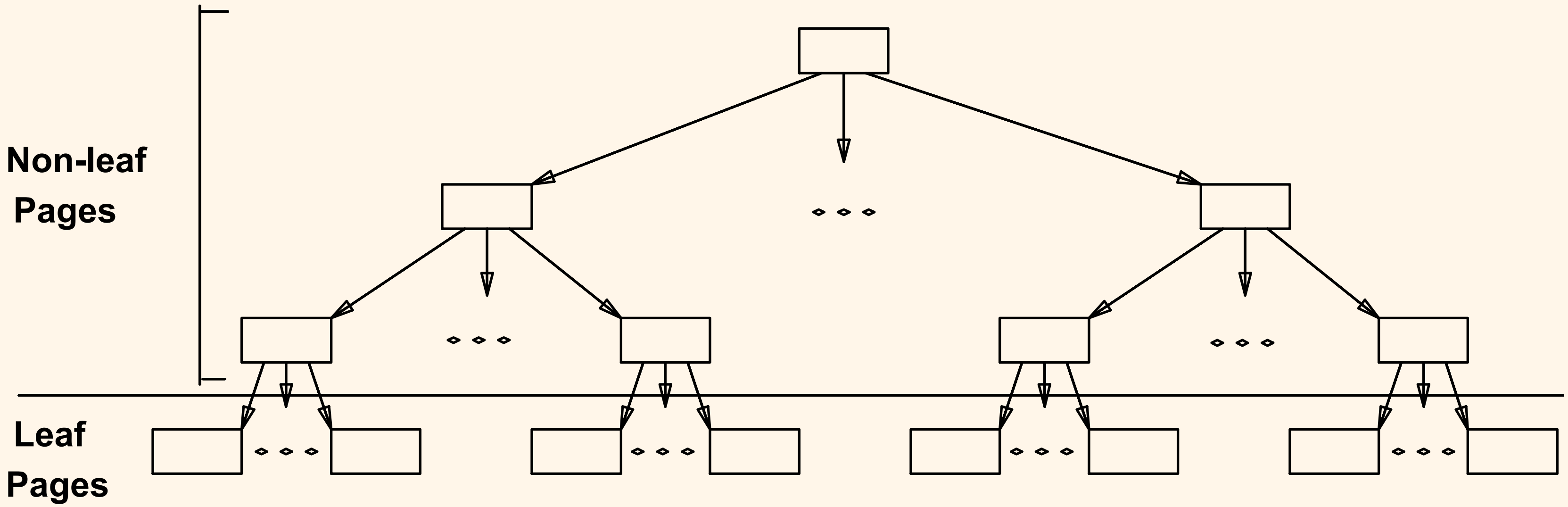
- Simple idea: Create an "index file"
  - What is search cost if each index page has F entries?



- *Can do binary search on (smaller) index file!*

ISAM

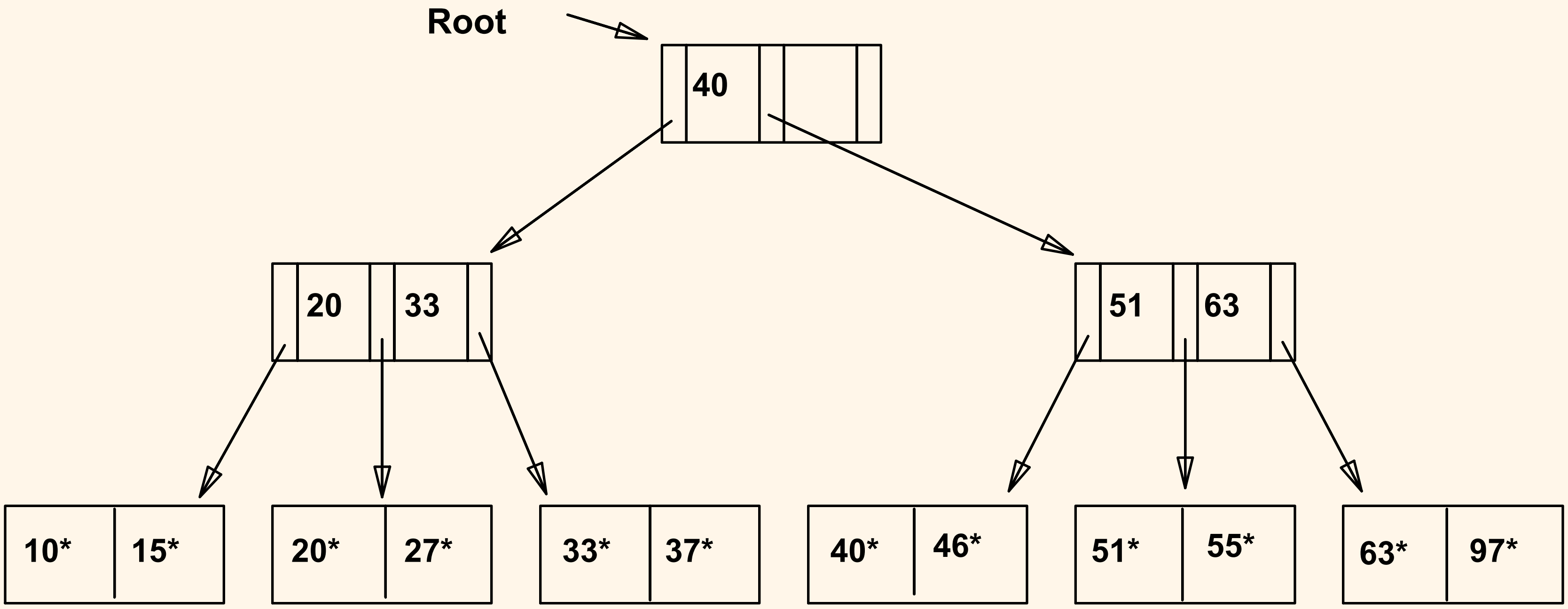
- Index file may still be quite large. But we can apply the idea repeatedly!



- *Leaf pages contain data entries.*

Example ISAM Tree

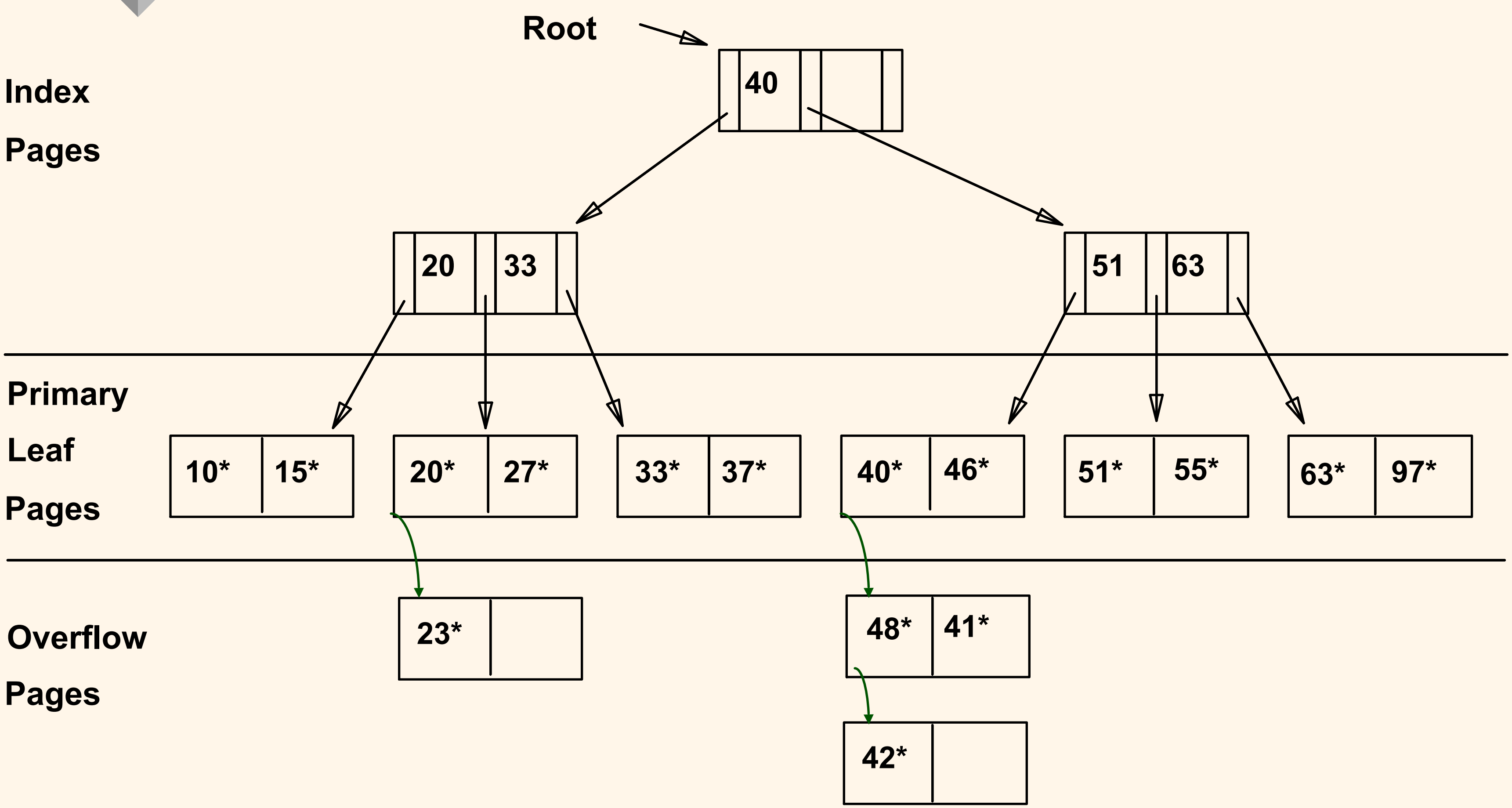
- Each node can hold 2 entries (one node = one page)
  - What is search cost if have N data entries, each leaf node can hold L data entries and each index node can hold F index entries (i.e. F+1 pointers)?



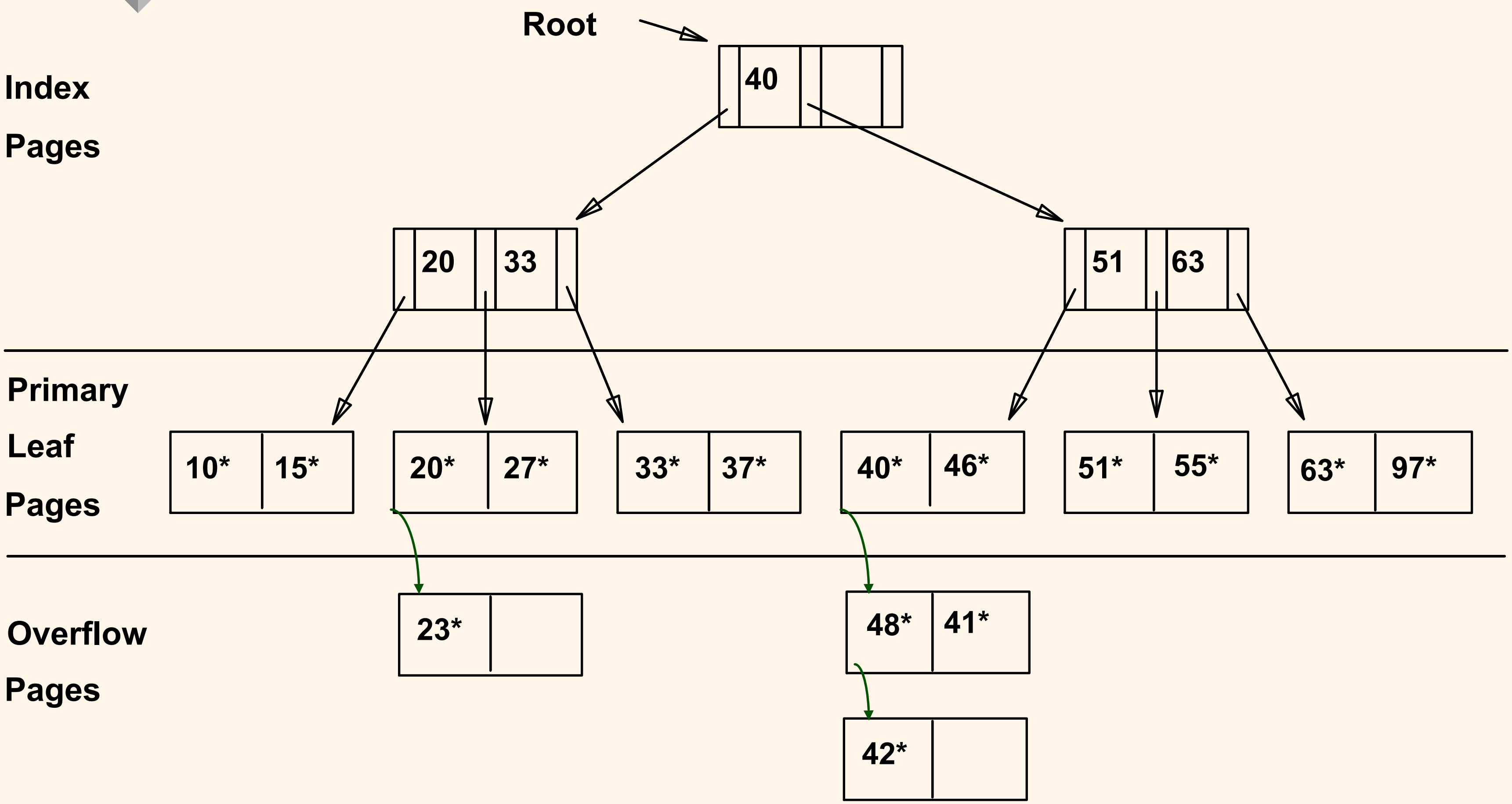
Overflow pages

- The ISAM tree is static – index nodes never change after construction
- If we have lots of new data entries being inserted, may need to add overflow pages at the leaves

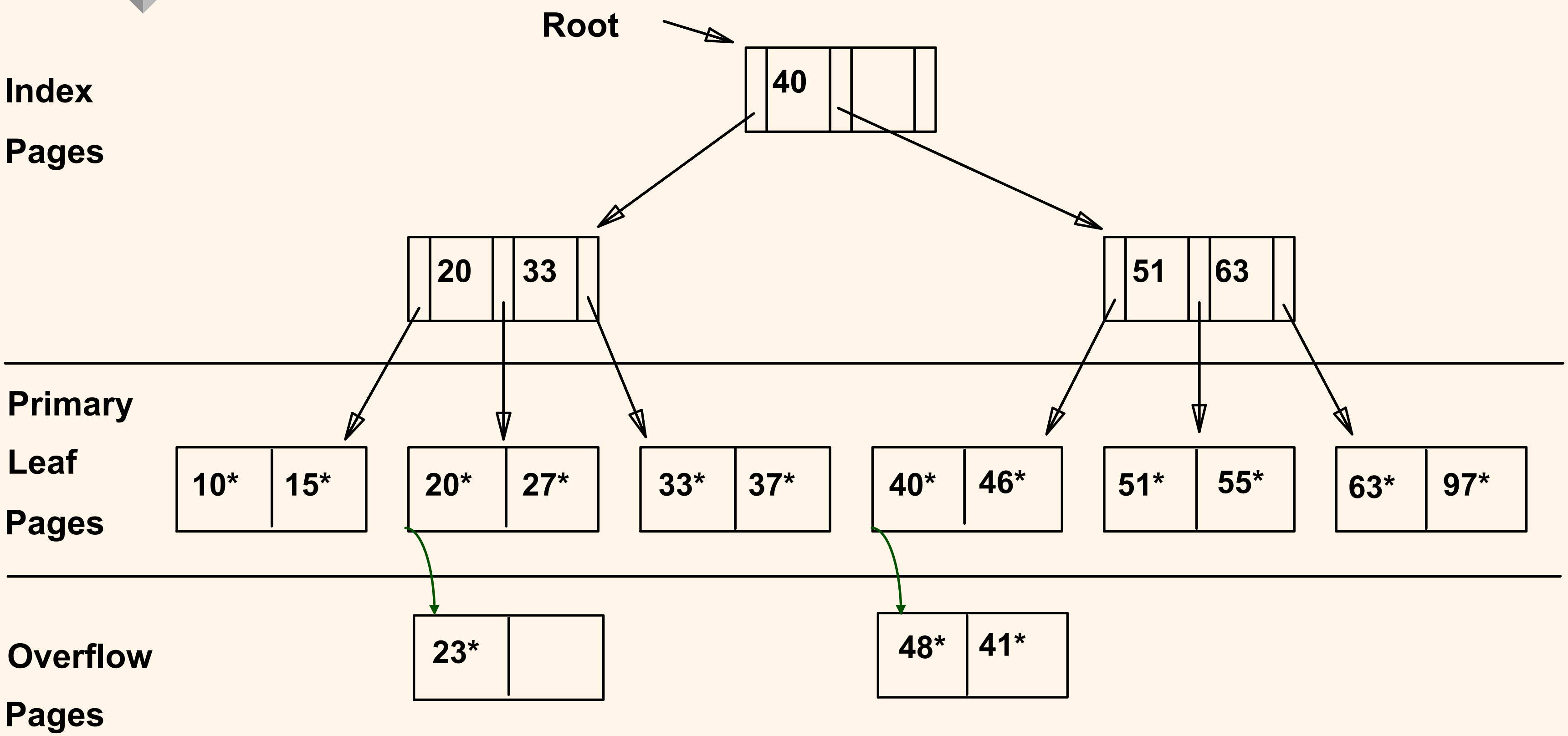
After Inserting 23*, 48*, 41*, 42* ...



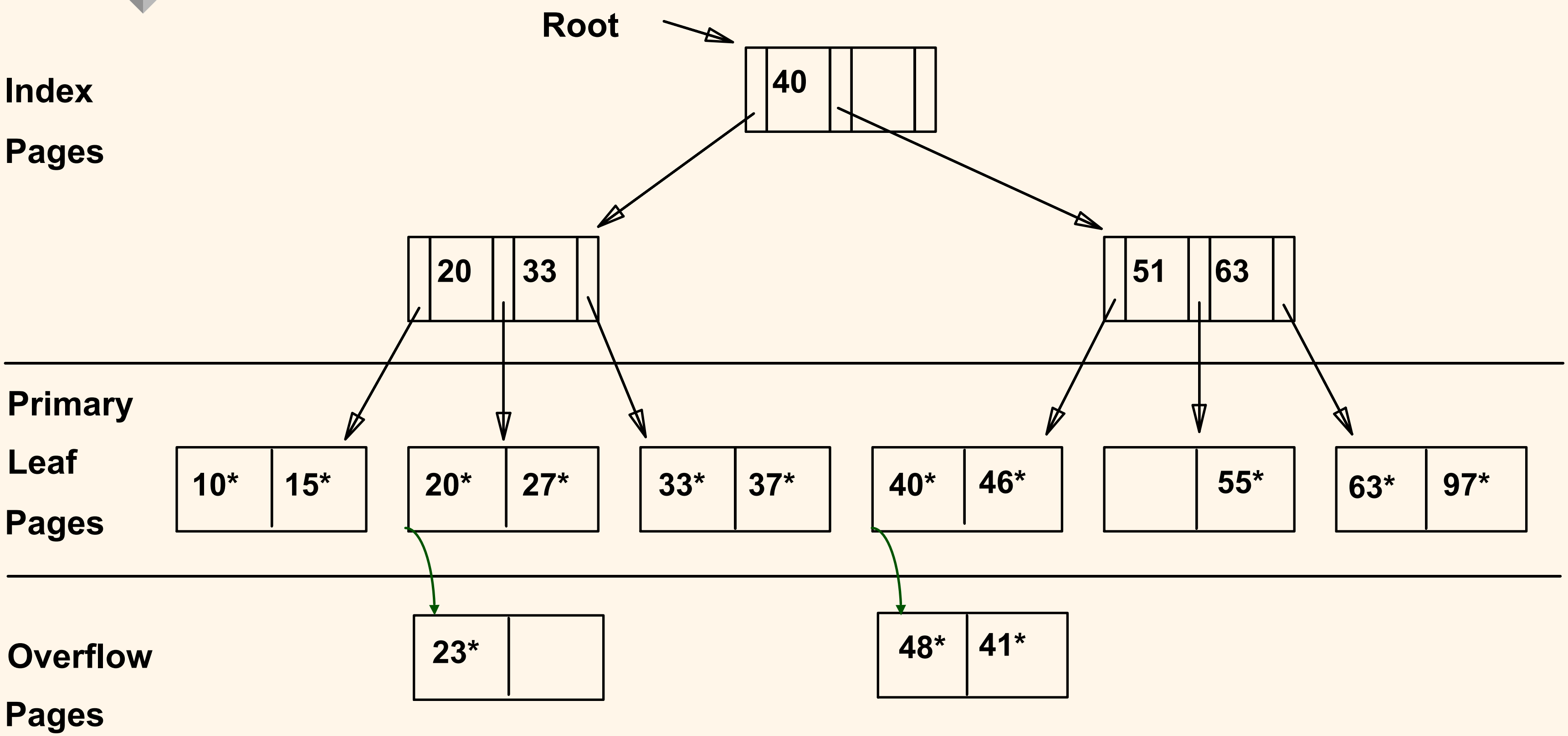
… Then Deleting 42*



... Then Deleting 51*



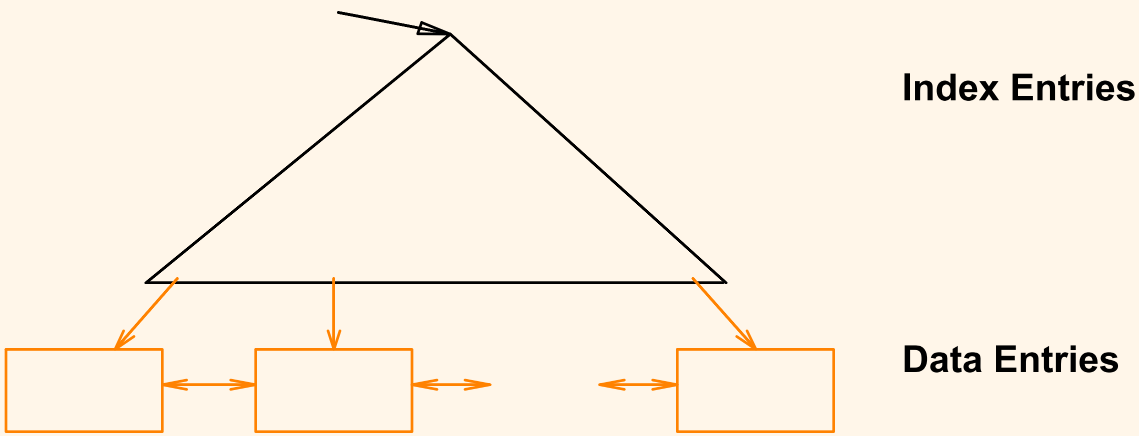
After Deleting 42* and 51*



Note 51 appears in Index Page but not in Leaf pages!

B+ Tree: The Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Fanout = # children per node
- Minimum 50% occupancy (except for root). Each node contains **d** <= $\underline{m}$ <= 2**d** entries. The parameter **d** is called the *order* of the tree.

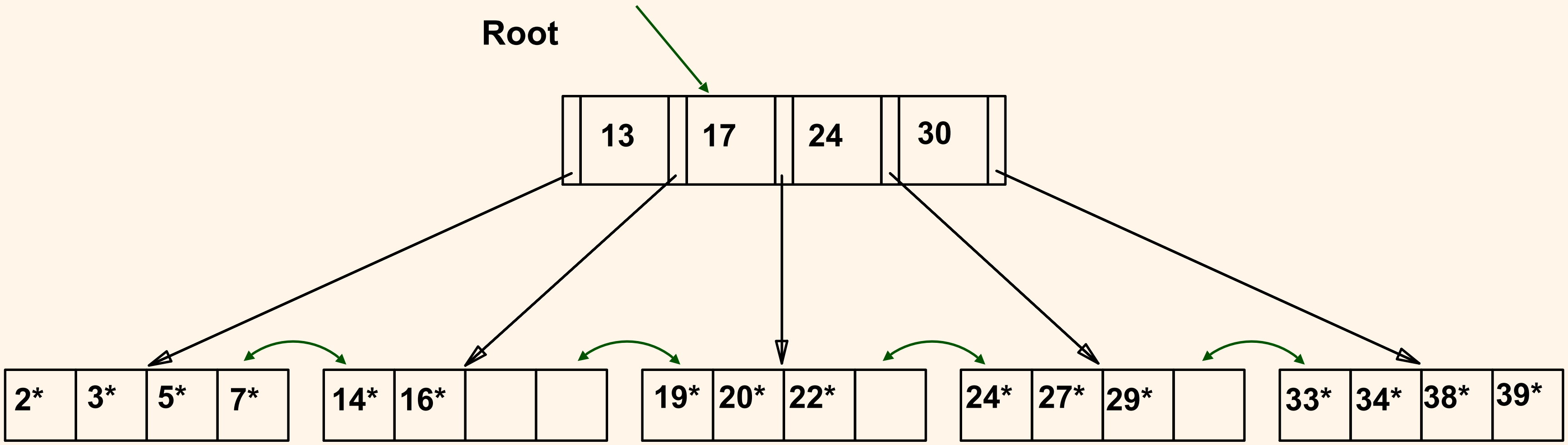


B+ Trees in Practice

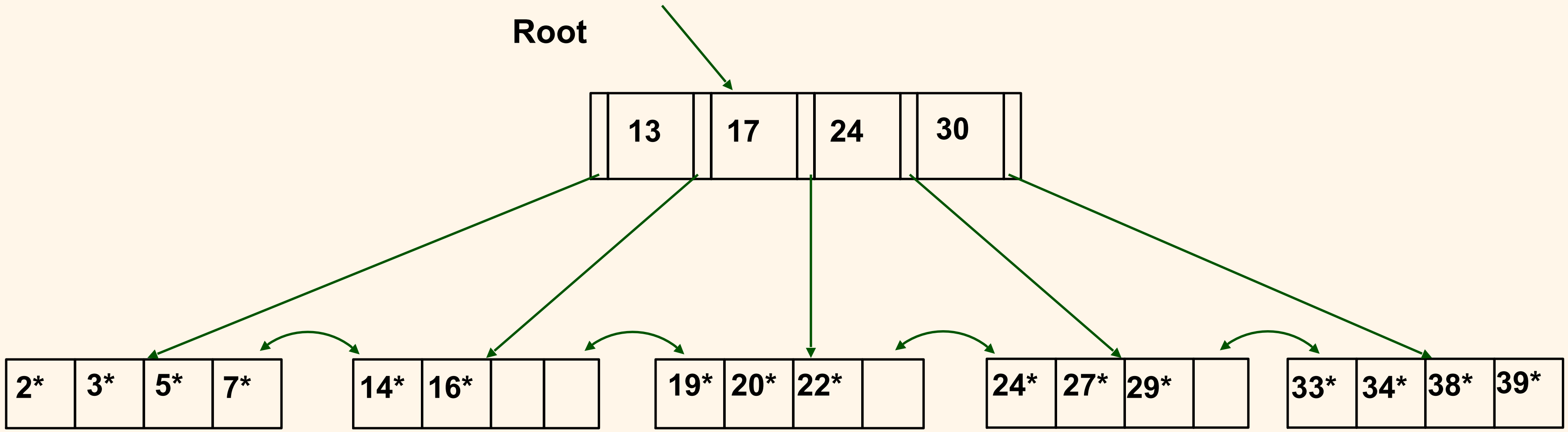
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Large fanout keeps tree as "short" as possible:
  - If root has 133 children,
  - it can have 17,689 grandchildren
  - 2,352,637 great-grandchildren
  - 312,900,721 great-great-grandchildren
- Serialized (stored on disk) so there is <u>one node per page</u>

Example B+ Tree

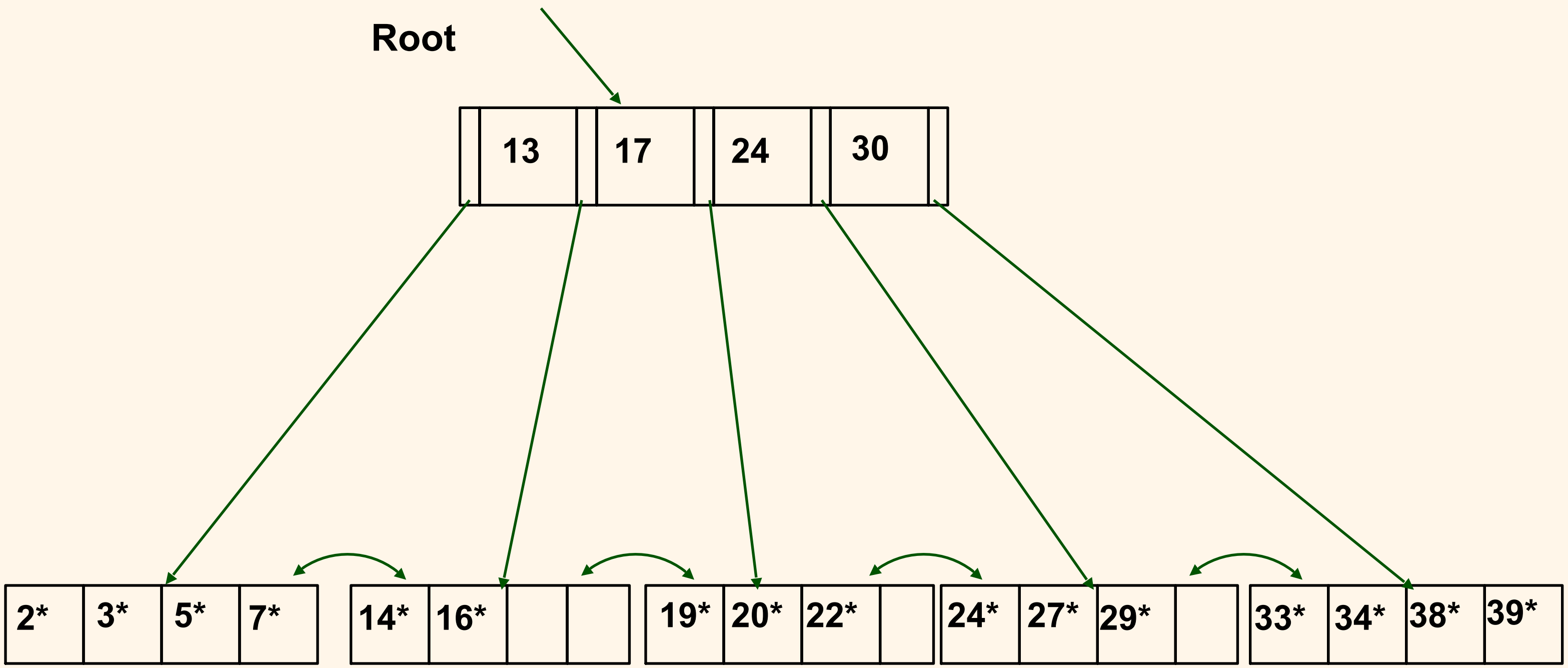
- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



Inserting 23*

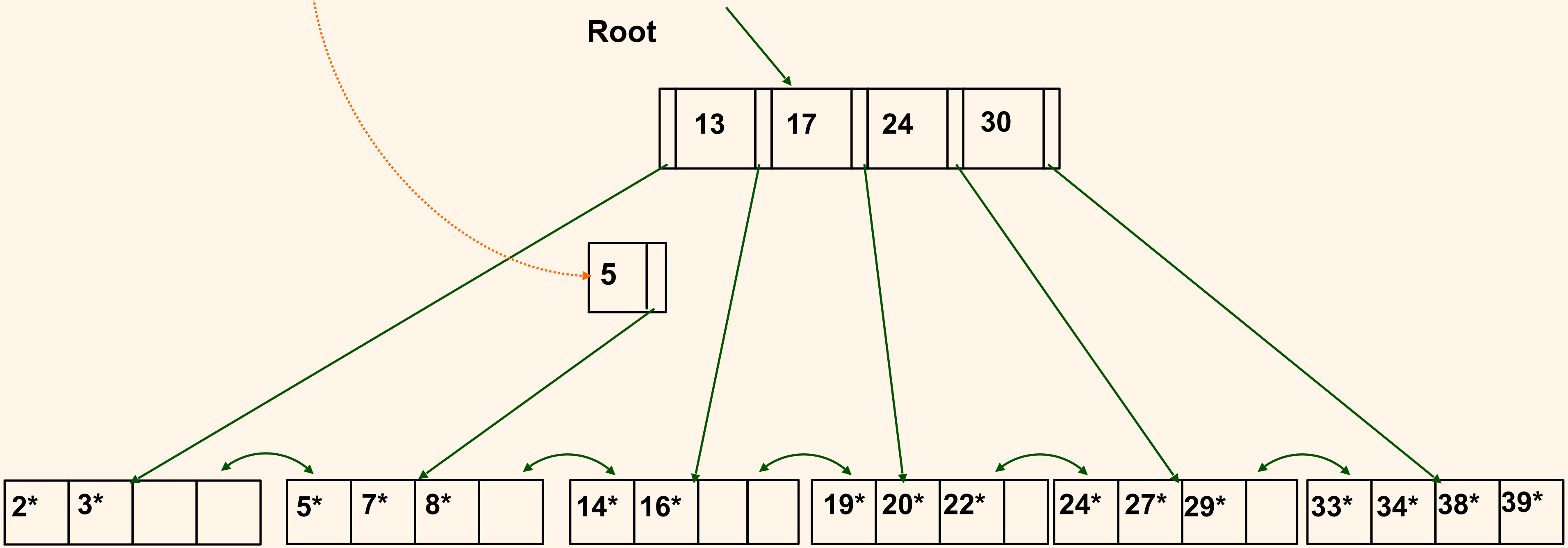


Inserting 8 ...*



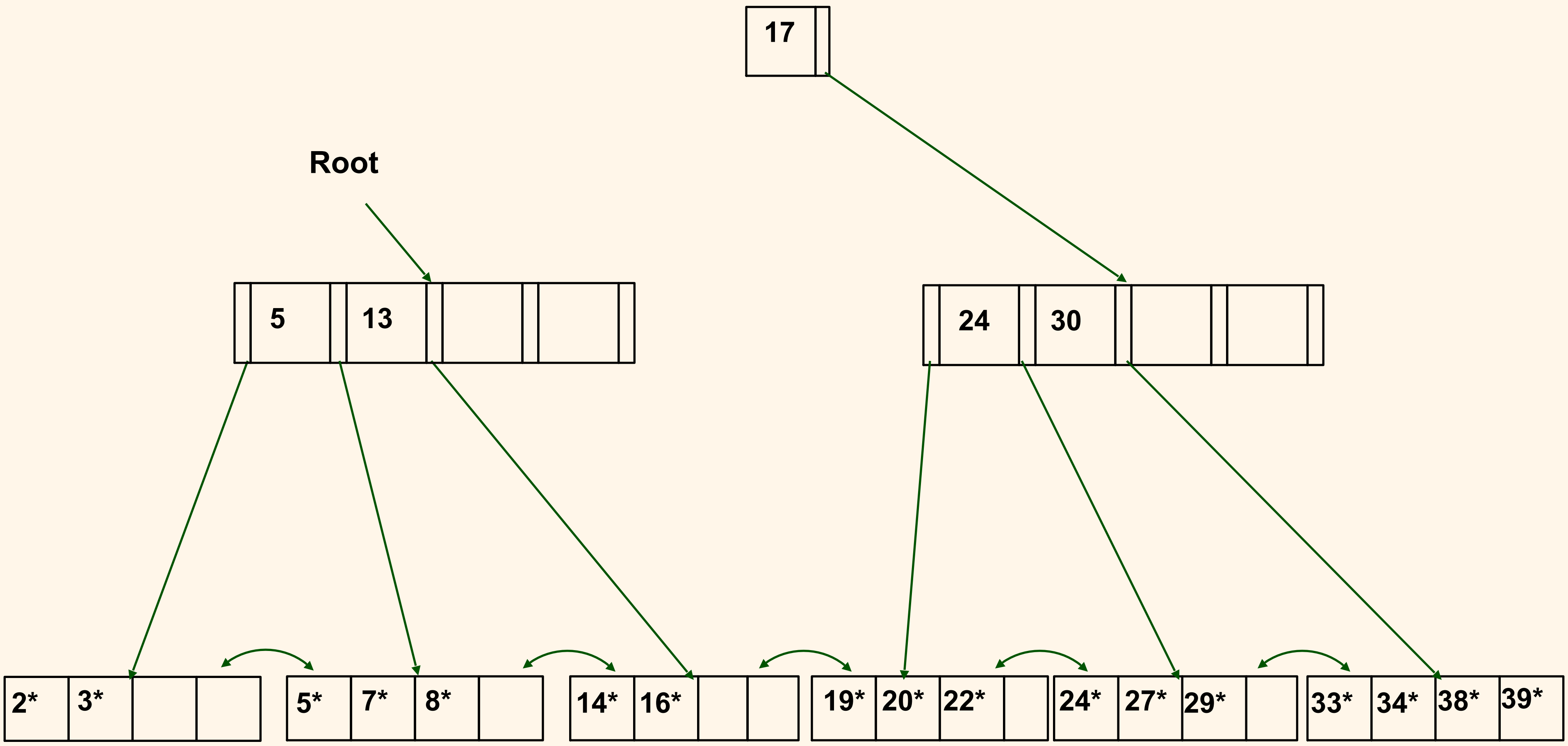
Inserting 8 ...*

Entry to be inserted in parent node
(Note that 5 is copied up and
continues to appear in the leaf)



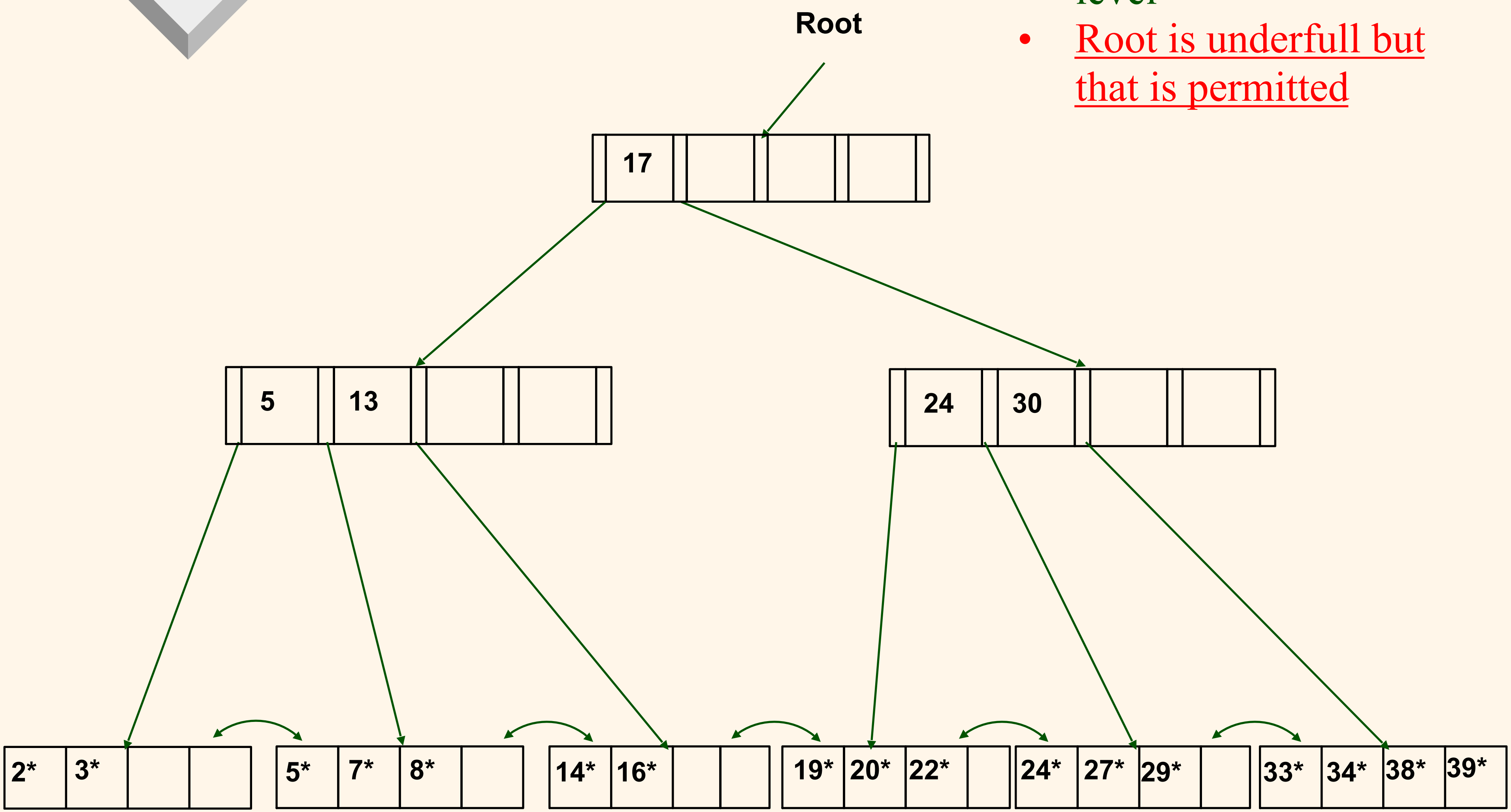
Inserting 8*

Entry to be inserted in parent node (Note that 17 is pushed up and only appears once in the index. Contrast this with leaf split)

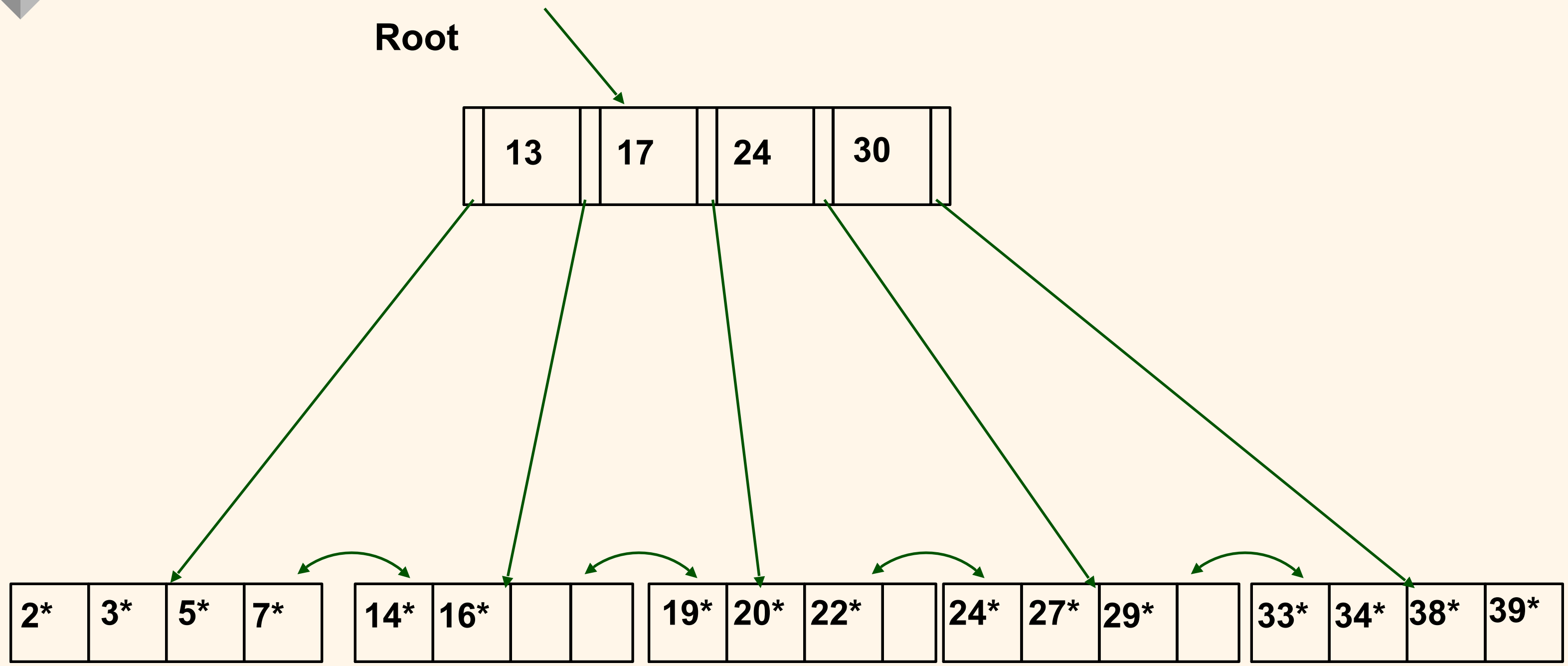


After Inserting 8*

- Tree grew by one level
- Root is underfull but that is permitted

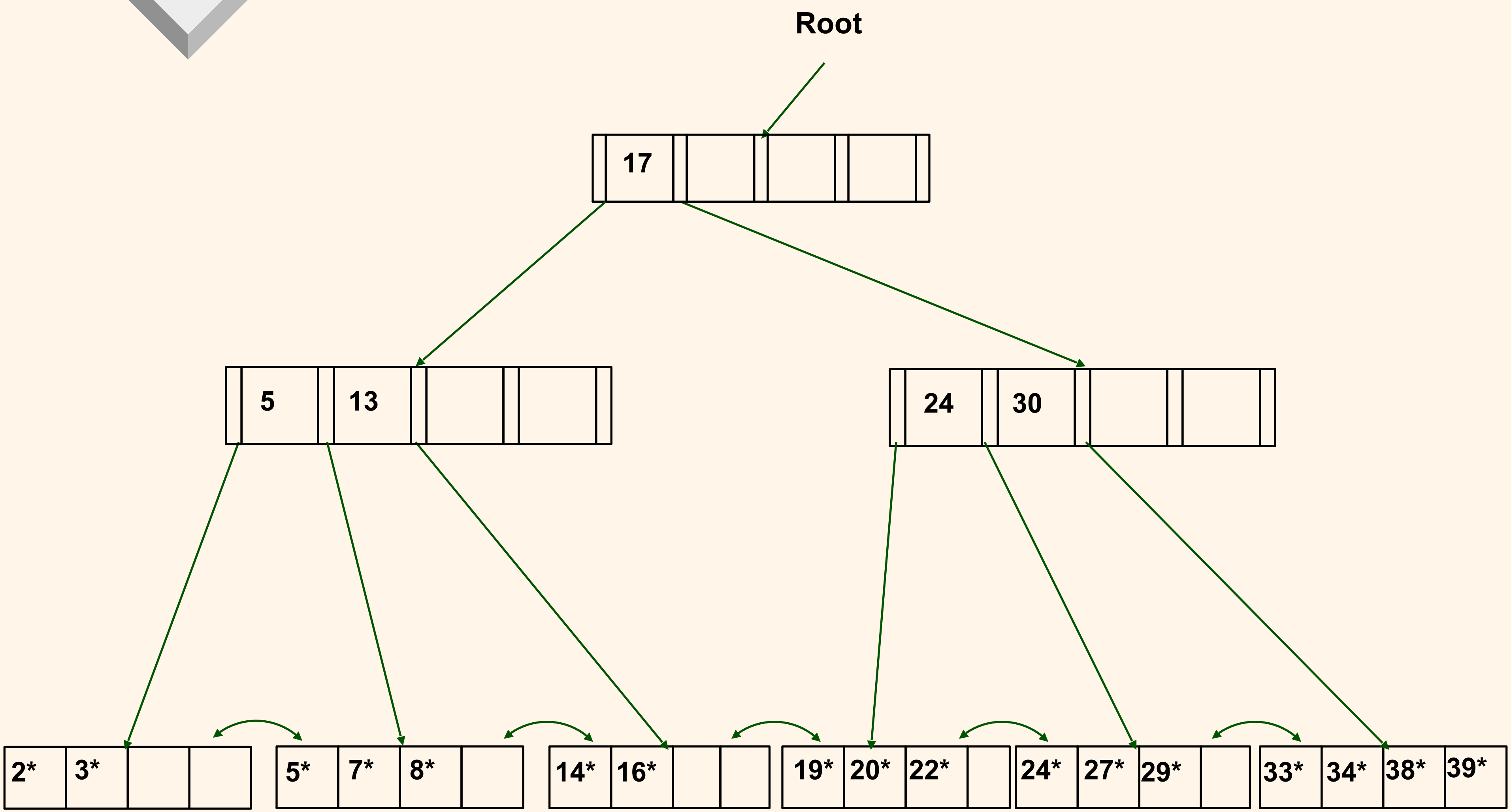


Inserting 8* ...

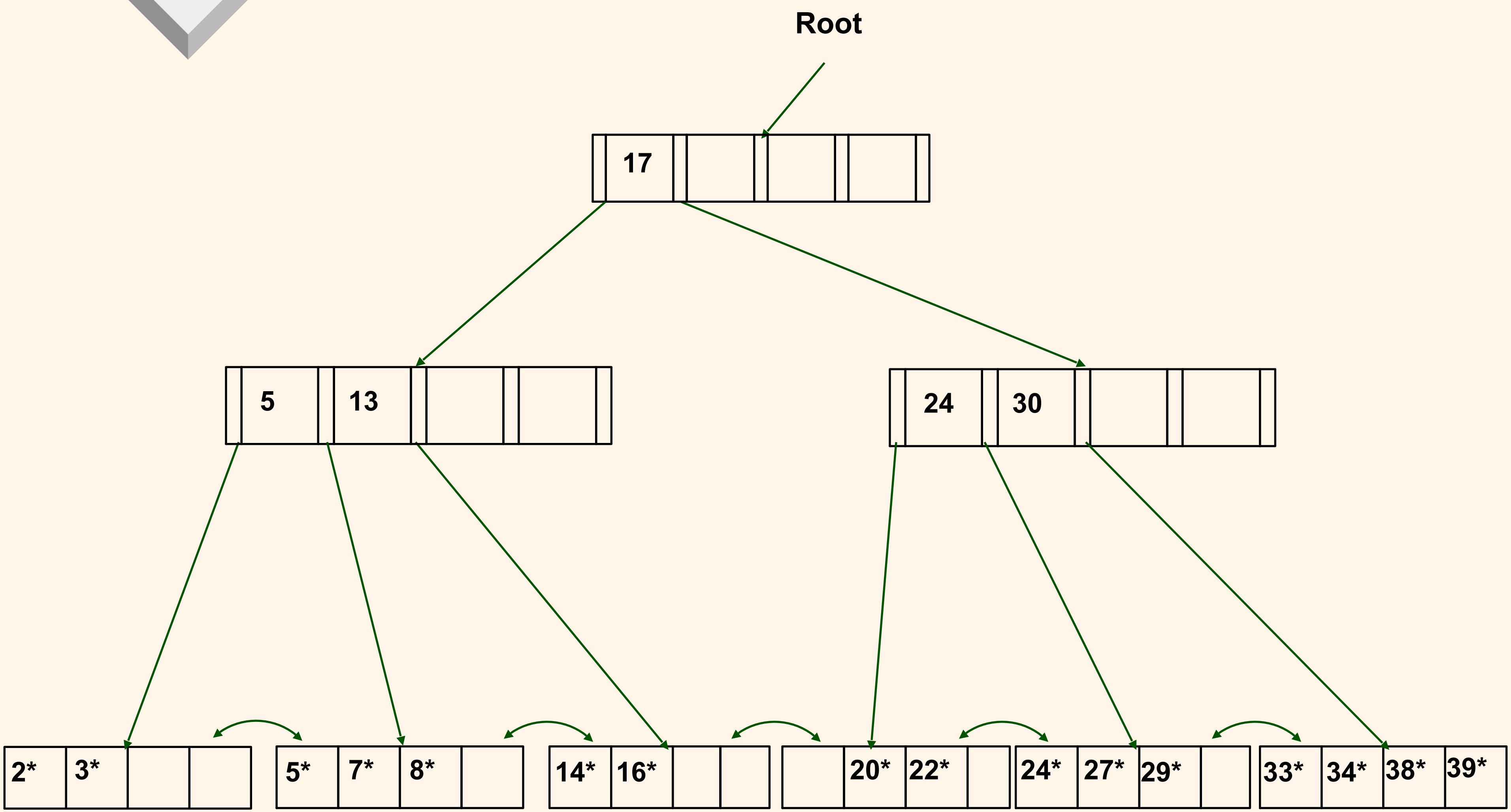


- In this example, could have "redistributed" to sibling instead of splitting

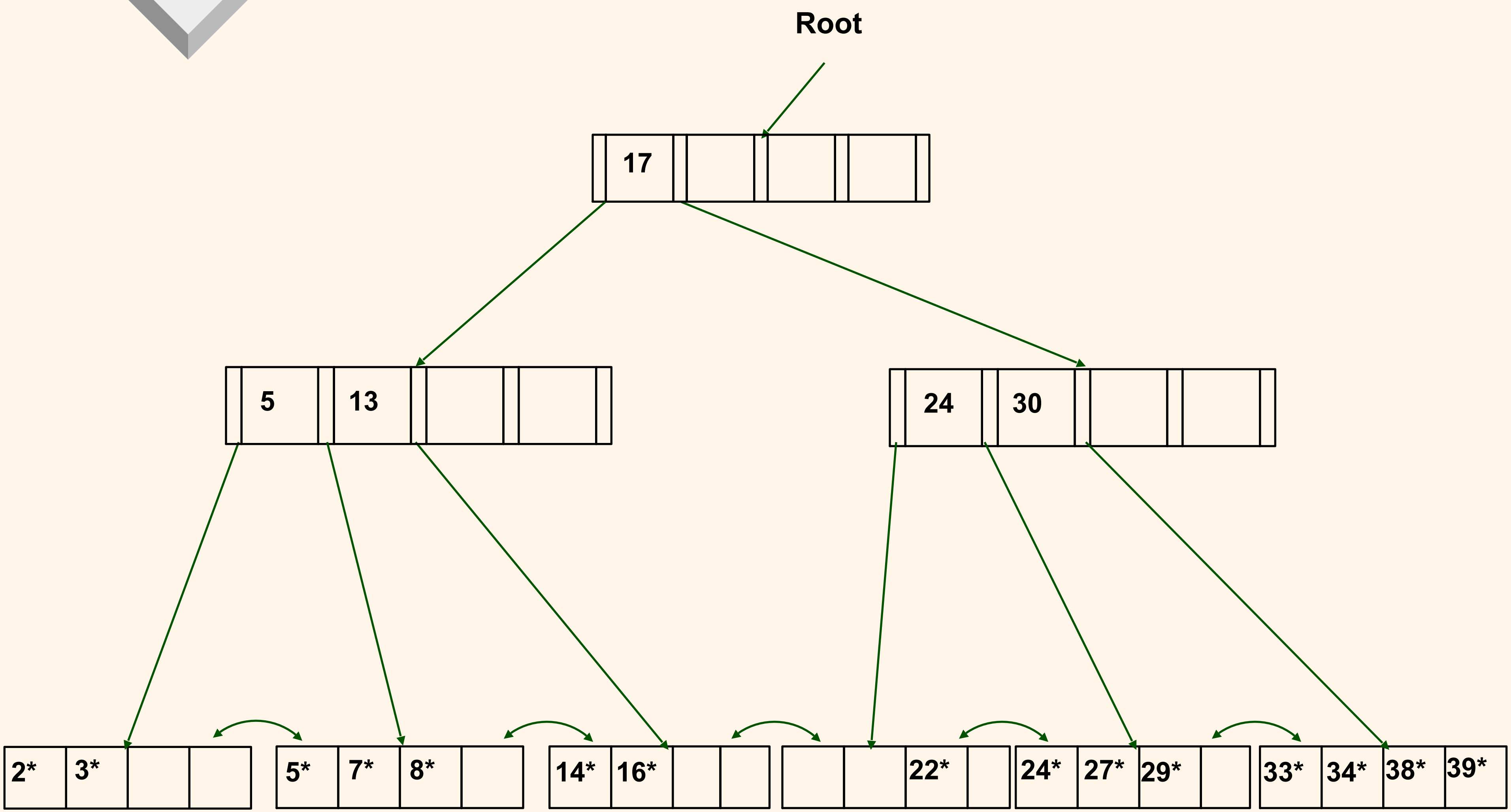
Deleting 19* ...



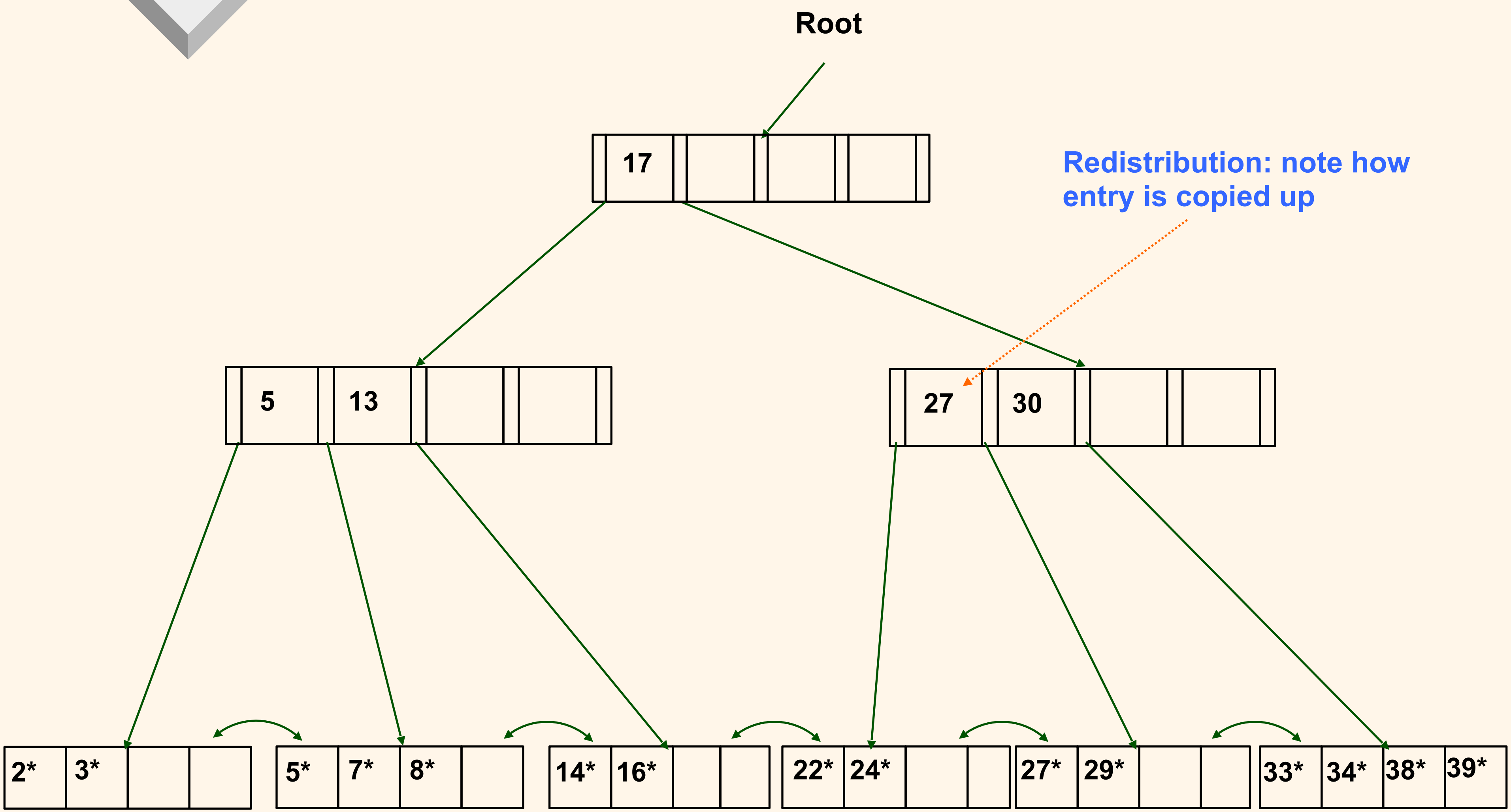
Deleting 20* ...



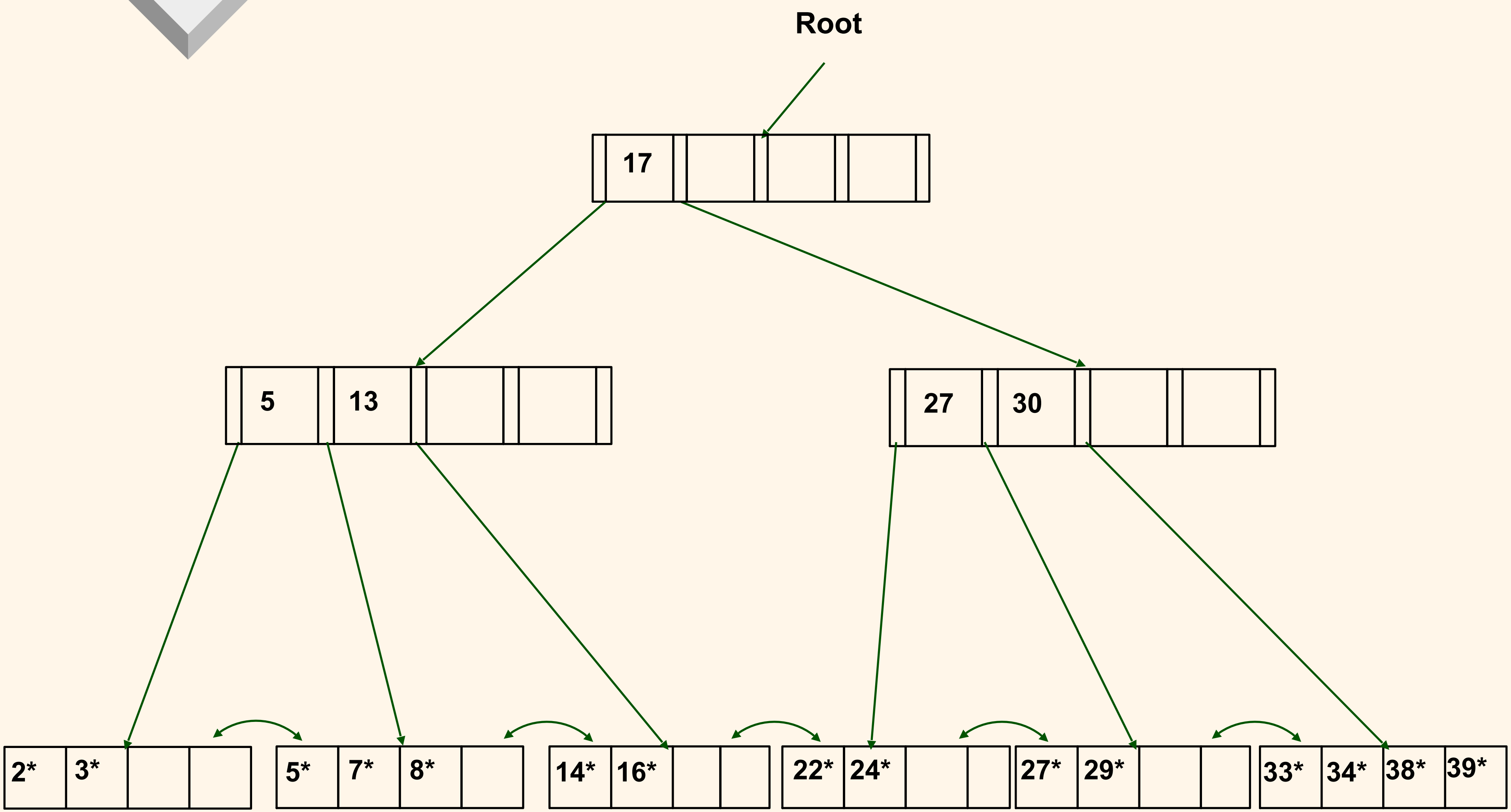
Deleting 20* ...



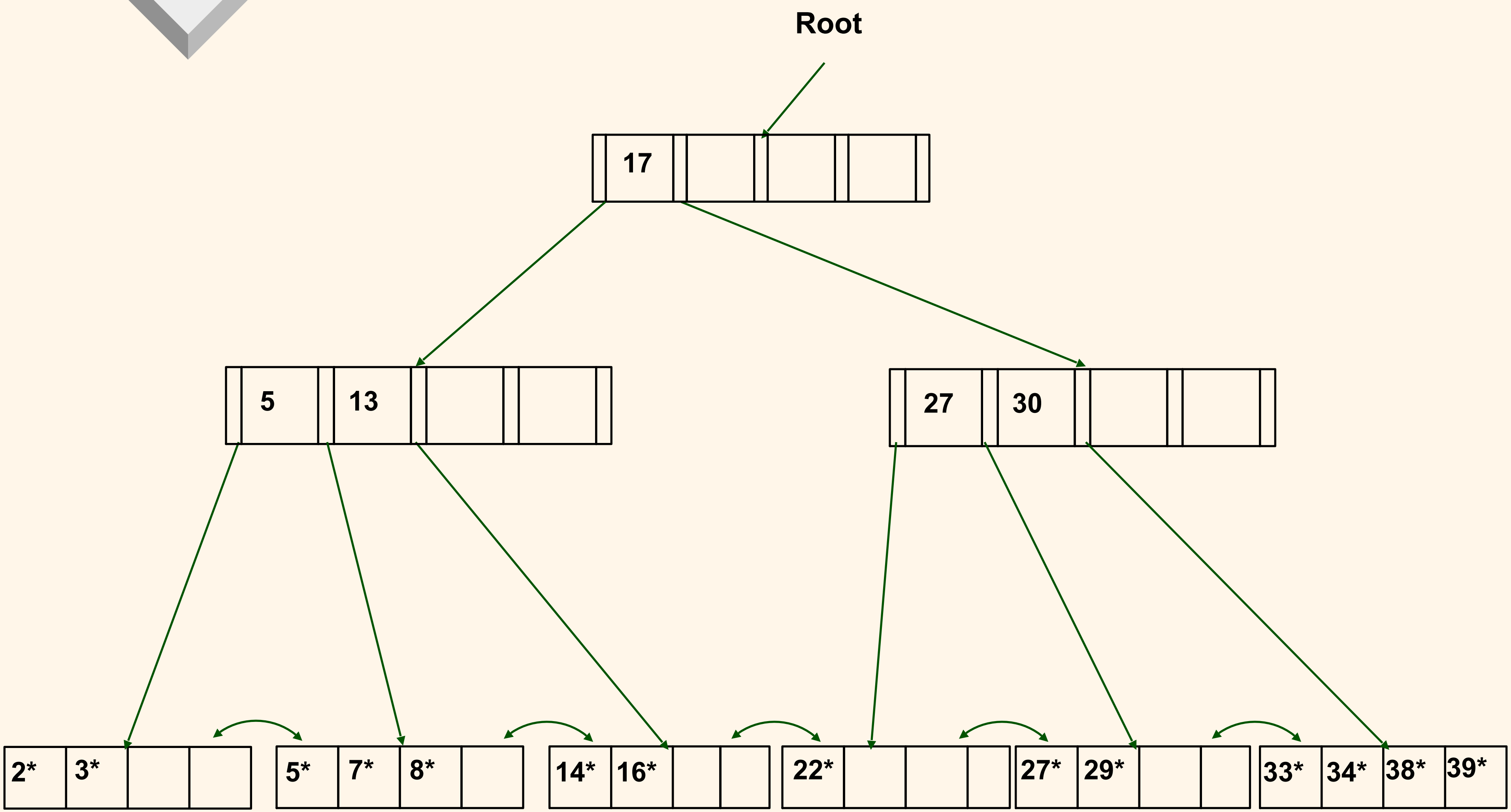
After Deleting 20*



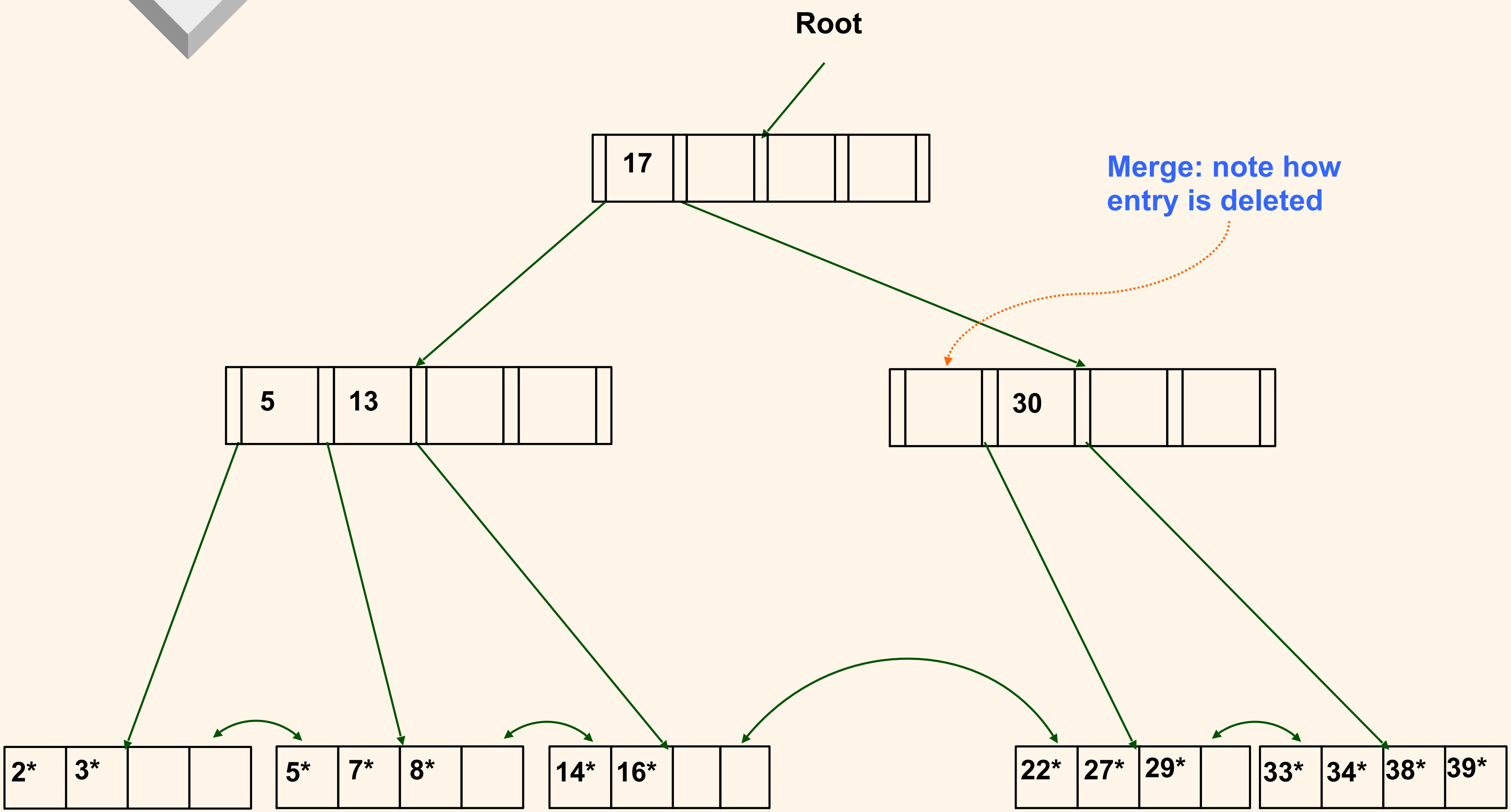
Deleting 24* ...



Deleting 24* ...

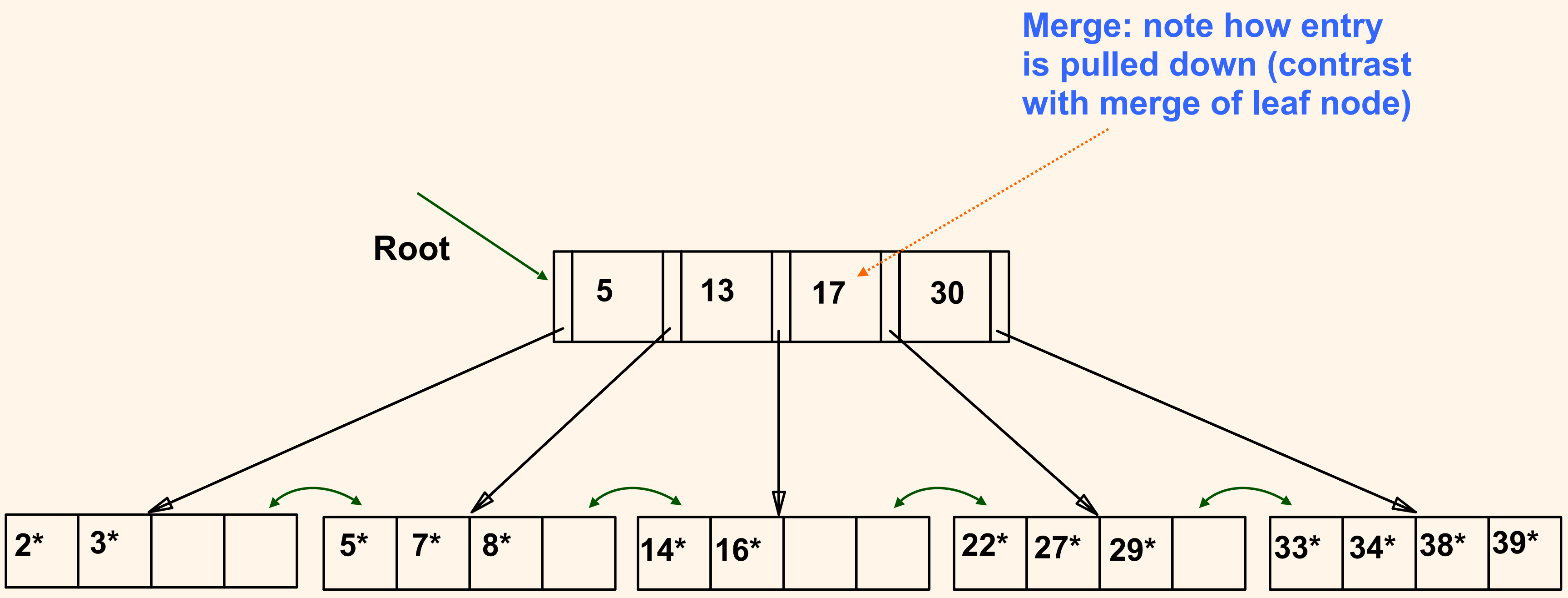


Deleting 24* ...



Root
17
Merge: note how entry is deleted
5
13
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

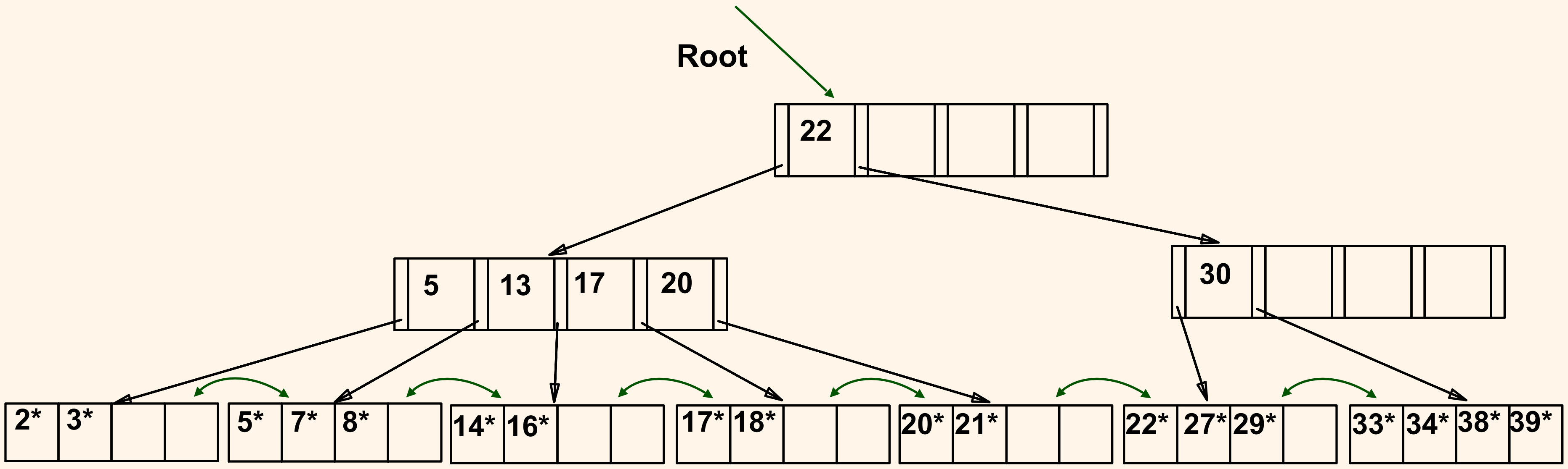
Deleting 24* ...



Merge: note how entry is pulled down (contrast with merge of leaf node)
Root
5
13
17
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

Example of Non-leaf Re-distribution

- This is a tree part-way through a deletion
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Entries are re-distributed by *pushing through* the splitting entry in the parent node.
- Suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.